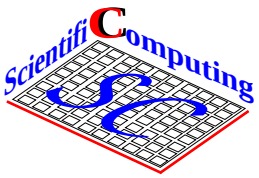
Boris Bügling, Martin Krosche

Coupling the CTL and MATLAB

Informatikbericht Nr.: 2007-03

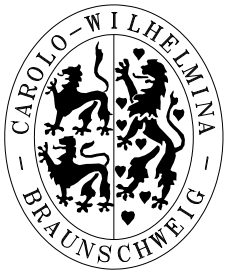


Coupling the CTL and MATLAB

Boris Bügling, Martin Krosche
Institute of Scientific Computing
Technical University Braunschweig
Brunswick, Germany

Informatikbericht Nr.: 2007-03

September 2007



Coupling the CTL and MATLAB

Boris Bügling, Martin Krosche
Department of Mathematics and Computer Science
Technical University Braunschweig
Brunswick, Germany

Informatikbericht Nr.: 2007-03

September 2007

Location
Institute of Scientific Computing
Technical University Braunschweig
Hans-Sommer-Strasse 65
D-38106 Braunschweig

Postal Address
Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
D-38092 Braunschweig
Germany

Contact

Phone:	+49-(0)531-391-3000
Fax:	+49-(0)531-391-3003
E–Mail:	wire@tu-bs.de
www:	http://www.wire.tu-bs.de

Coupling the CTL and MATLAB

Boris Bügling Martin Krosche
Institute of Scientific Computing
Technische Universität Braunschweig
Brunswick, Germany
September 2007

Contents

Listings

Abstract

The programming language MATLAB is widespread in the field of scientific computing. In some cases it may be reasonable to use external software, which is to be embedded into or invoked from MATLAB code. Interfacing Java code from MATLAB is directly available. The MATLAB compiler (MCC) and the MATLAB executables (MEX) enable the coupling to C/C++ code. If the desired software system is highly hybrid and distributed, that means many different pieces of code speak to each other possibly over network, the approach can be cost-intensive. The Component Template Library (CTL) is a middleware which supports the coupling of C/C++, Fortran, Java or Python code. In this paper we present how to couple CTL additionally with MATLAB.

1 Introduction

In the field of scientific computing many programming languages are common, like C/C++, Fortran, Java, Python or MATLAB. To reduce development time it is desirable to reuse code. Coupling code written in different programming languages in a manual manner can be really expensive. Thus it is reasonable to use special code, which relieves one of the coupling and supports many programming languages. In addition to that, it is desired to allow distributed and parallel computing. For the latter case a middleware is commonly used.

The *Component Template Library* (*CTL*) [4] is such a middleware, which enables coupling different pieces of code as mentioned above. It is a C++ template library for realising distributed software systems. In this context a CTL component is a software, which is specified by an interface, called the *Component Interface* (*CI*). The CTL C++ library allows to implement C/C++ and Fortran components without using wrapper code. To write components in Java the correspondent *CTL4j* [1] was developed.

This paper deals with coupling the CTL and MATLAB code in both directions. We confine ourself to C++ and Java. Chapter 2 introduces the reader to MATLAB executables (MEX) and the MATLAB Compiler (MCC). How to build a MEX-file from a CTL component is described in chapter 3, and how to call a CTL4j component is presented in chapter 4. The following chapter 5 considers the realisation of MATLAB code as a CTL component by using the MCC. Basics of the CTL are addressed in the CTL manual [4] and the CTL4j project work [1]. In order to familiarise the reader with both the C and C++ API of MATLAB, the MEX chapter uses the former and the MCC chapter the latter.

2 Introduction to the MATLAB C/C++ API

In this chapter we give some annotations onto MATLAB's MEX and MCC. We consider a known bug in some MCC versions and how we handle it. Additionally we present the possible MATLAB runtime environments.

2.1 MATLAB's C/C++ API

When interfacing with MATLAB, one first needs to know how to convert between the standard types of the programming language and the types MATLAB uses internally and exports via its API. The C++ class `mwArray` realises a mapping between MATLAB's intrinsic types and C++. For C programmers, there is the type `mxArray`, which basically provides the same functionality in C-style (no object-orientation).

All the data passed to or gotten from MATLAB needs to be converted to the type `mwArray`, which can hold a single value of the fundamental types integer (signed or unsigned and 8, 16 or 32 byte long), single (single precision floating point number) or double. It can also hold a string or a multi-dimensional array. For example, one can use a constructor which takes the number of rows and columns, plus the type of the fields (e.g. `mxDOUBLE_CLASS` for double) and the domain (real or complex), to get a matrix-type, which interfaces with MATLAB. The method `SetData()` can set all the fields in a `mwArray` from a C++ standard array of the corresponding type.

Of course, this approach decreases usability and portability. Therefore additional wrapper code was developed and integrated into some code-generators, which handle all the conversion from C++ to MATLAB types and vice versa without user-intervention. More on this is considered in the subsequent chapters.

2.2 Introduction to MEX

To make third-party code available to MATLAB, one needs to compile a shared library, called a MEX-file. This library is executable from within MATLAB like a build-in function or script. MATLAB ships with a compiler, also called *mex*, which generates such a library from a C code.

MATLAB's API contains external interface functions, which provide mechanisms for converting data between native C and MATLAB C types, as well as making built-in functions available to the external C code.

Usually, MEX is used to call compiled code with better performance from inside MATLAB. But in our case, it provides a means to integrate CTL components into MATLAB and access to remotely available CTL resources. Chapter 3 will introduce all the concepts of writing such a MEX-file manually and automatically.

2.3 Introduction to MCC

The MCC is a wrapper around GCC, which generates C++ wrapper code around one or more existing MATLAB M-functions. This gets compiled and linked to MATLAB's own shared libraries to produce native executables or shared libraries, which can be used without starting the whole MATLAB environment. The process for reaching the MCC generated shared libraries can be separated into four steps. Consider these steps for resulting C++ code by looking at two M-files *func1.m* and *func2.m*. The first step contains the generation of the C++ wrapper code done by the MCC. This results in some wrapper code files, especially *libfuncs.h*, *libfuncs.cpp* and *libfuncs_mcc_component.c*. The two implementation codes need to be compiled in the second and third steps. In the fourth step everything is linked together to the desired shared library. Of course the above header as well as the MATLAB intrinsic header *mclcppclass.h* need to be included in the calling code and the library needs to be linked after compiling the callee. Notice that the wrapper code deals with `mwArray` (see section 2.1) objects directly. We do not want to handle these types inside the caller due to more portability, so we convert them.

The MCC V4.3 (R14SP3) contains a known bug in terms of matrix handling. The matrix passed from C++ to MATLAB is interpreted in different ways. The dimensions of both matrices — the MATLAB and the C++ ones — are maintained, but the entries are written row-wise to the columns. A demonstration is given in listing 1. In the reverse way, when a matrix is given from MATLAB to C++, the dimensions of both matrices are also maintained, but now the columns are written to the rows, see listing 2.

```
# C++:
       1     2     3
       4     5     6
# MATLAB:
  A =
       1     3     5
       2     4     6
```

Listing 1: MCC bug: rows are written to columns.

```
# MATLAB:
  B =
       2     4
       6     8
       10    12
# C++:
       2     6
       10    4
       8     12
```

Listing 2: MCC bug: columns are written to rows.

As vectors are treated as 1xn or nx1 matrices by MATLAB the above bug affects vectors too. This does not have an effect, as the dimensions are maintained and only one row or column exists.

Additionally we want to mention, that STL C++ vectors are not specified in being a row or column vector in contrast to MATLAB. When passing a vector to MATLAB this plays a role as inside the MATLAB code this separation is required typically. For these cases one could introduce a flag for C++ vectors indicating if the vector should be a row or a column vector.

We implemented a C++ parser class `mwArray_prs`, which converts the most interesting MATLAB specific C++ types to standard C++ fundamentals and containers, can handle row and column vectors and optionally the above bug. Additionally, there is an automatic code-generator, which does not support all of these functionalities, see chapter 5.

2.4 MATLAB vs. MCR

There are two possibilities running a CTL component interfacing a MATLAB code. One way is to use the common MATLAB installation. To do so, typically the MATLAB license server is contacted, and thus one license is taken per machine involved. That is why we do not recommend this approach.

An alternative way is using the MATLAB Component Runtime (MCR) [3]. The MCR is a virtual machine, which can execute wrapper code generated by the MCC. To get the machine dependant MCR, the MATLAB intrinsic script *toolbox/compiler/buildmcr* needs to be invoked inside the MATLAB environment. It results in the compressed package *MCRInstall.zip* containing the required MCR data. This package can be extracted to an arbitrary path on the desired machine. As the MCR does not contact MATLAB's license server, this is the cheaper and thus recommended approach, which is used in this paper.

More information on using MCR can be found in the MATLAB documentation and on the corresponding website [3].

3 How to Use CTL/C++ Components from MATLAB

A MEX-file is a dynamically linked library containing only one subroutine, which can be used from within MATLAB just like built-in functions or scripts. This section describes how to write such a function manually first. After that we present a code-generator, which parses a CI and generates C++ code calling the CTL component. This code can be compiled and linked into a MEX-file.

3.1 Manual Approach

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <mex.h>

#include <ci/add.ci>

extern "C"
{
void mexFunction (int nresults, mxArray **arrresults, int nargs,
                const mxArray **arrargs)
{
        int i = 0, j = 0;
        if (nargs > 0)
                i = int(mxGetScalar(arrargs[0]));
        if (nargs > 1)
                j = int(mxGetScalar(arrargs[1]));

        const char *loc = "/home/neocool/test/ctl-gcc3.3/examples1/add/linux-gcc/
            add.exe -l pipe";
        ctl::link lnk(loc);
        if (!lnk)
        {
                std::cerr <<"Could not create link -> " << loc << "\n";
                return;
        }

        AddCI add(lnk);
        if (!add)
        {
                std::cerr <<"Could not create addCI -> " << loc << "\n";
                return;
        }

        try
        {
                int res = add.add(i, j);
                std::cerr << loc << " -> Add: " << i << " + " << j << " = " <<
                    res << "\n";
        }
        catch (ctl::exception &e)
        {
                std::cerr << e <<"\n";
        }
}
}
```

Listing 3: Example mexFunction

A MEX-file has the file extension *.mexglx*. The base name of this file is used as the function name by MATLAB. Consider a simple C++ class `Add` performing an addition of two integers via method `add()`. The CI `AddCI` taken from the CTL examples realises the corresponding CTL component. It contains a constructor and the method `add()`. The source code of the desired *.mexglx* file, which interfaces the component, is shown in listing 3. MATLAB calls the component via the

C function `mexFunction()`. It needs to have the signature seen in the listing, containing the four arguments:

1. The number of results expected by the caller,
2. the array to which those results are passed,
3. the number of arguments,
4. the array of those arguments.

MATLAB's *mex.h* header defines the type `mxArray` (see chapter 2.1). It also defines helper functions for converting this type to C types. For instance function `mxGetScalar()` retrieves a double from a `mxArray`. This means, `mexFunction()` has three tasks to fulfill: converting the arguments to C types, calling one or more functions provided by CTL components and finally converting the results back into MATLAB types. One can compile a MEX-file by using the following command (here, *add.cpp* contains the `mexFunction()`):

```
$MEX_HOME/mex add.cpp
```

This creates a file called *add.mexglx*. If it is found by MATLAB (by either being in its path or in the current working directory), it can be called as `add()`.

Essentially, both the conversions as well as the CTL function call itself can be derived from the CI and therefore automatically generated, see next section. In contrast to our example, of course the code of the `mexFunction()` could be more complex for instance involving multiple components, but the interface to MATLAB is always one function.

3.2 Automatic Approach

The work needed to call CTL functions from MATLAB can be automated. This section introduces an automatism for CIs containing a library with a single function, as this directly corresponds to the MEX-file philosophy (a MEX-file contains only one function). This is realised by the Python script *ctlmex.py*. The script takes a CI as input and delivers the source code of the `mexFunction()`, which enables MATLAB to call the function. To simplify the generated code, there are helper functions for converting from C types to MATLAB types and vice versa (for more details see file *mexhelper.cpp* of the example code). They are compiled and then linked to the resulting MEX-file. The following commands generate *add.mexglx*:

```
ctlmex.py add.ci >add.cpp
mex -c mexhelper.cpp
mex add.cpp mexhelper.o -o add.mexglx
```

Additionally, there is one pre-build MEX-file *use.mexglx*, which specifies the location of a CTL component, just like its counterpart in CTL/C++. Thus the call from MATLAB looks like listing 4.

```
use('AddCI', 'add/add.exe -l tcp')
res = add(3, 4)
```

Listing 4: Calling from MATLAB

3.2.1 Limitations

In the following we summarise the limitations of our presented automatism.

- There is currently no support for classes.
- The custom vector class of *mexhelper.cpp* treats all vectors as column vectors.
- Conversion from `mxArray` to `std::string` and vice versa is not yet implemented.
- Due to their nature as normal shared objects, any segmentation faults or similar problems in the *.mexglx* files will cause MATLAB to crash.

3.2.2 Example: Gaussian Elimination

By regarding a C++ implementation of the Gaussian elimination algorithm (see listing 5) we want to demonstrate the automatic conversion of vectors and matrices. After componentising the code, we use *ctlmex.py* and *mex* to couple with MATLAB.

```
#include <mexhelper.h>

static bool verbose = false;

template<class T> myVector<T> gaussElimination (matrix<T> A, myVector<T> b)
{
        if (verbose)
                std::cerr << "A:\n" << A << "\nb:\n" << b << "\n\n";
        int N = A.N();

        for (int j=1;j<N;j++)
                for (int i=j;i<N;i++)
                {
                        T m = A.get(i, j-1) / A.get(j-1, j-1);
                        A.set(i, A.get(i) - A.get(j-1) * m);
                        b[i] = b[i] - m*b[j-1];
                }

        myVector<T> x(N);
```

```
20          for (int i=0;i<N;i++)
21                  x[i] = 0;
22
23          x[N] = b[N] / A.get(N, N);
24          for (int j=N-1;j>=0;j--)
25                  x[j] = (b[j] - A.get(j, j+1, N) * x.get(j+1, N)) / A.get(j, j);
26
27          return x;
28  }
```

Listing 5: Gaussian elimination

The algorithm uses the matrix- and vector-classes from *mexhelper.h*, which provide the necessary addition and multiplication operators.

```
1  #ifndef __LINEQ_H__
2  #define __LINEQ_H__
3
4  #include <ctl.h>
5
6  #define CTL_Library LinEq
7  #include CTL_LibBegin
8
9  #define CTL_FunctionTmpl1 myVector<T>, solve, (matrix<T>, myVector<T>), 2, (T), 1
10
11 #include CTL_LibEnd
12
13 #endif
14 // vim: ft=cpp
```

Listing 6: Component interface

```
1  #define CTL_Connect
2
3  #include <gauss.h>
4  #include <lineq.ci>
5
6  void CTL_connect ()
7  {
8  #ifdef CTL_VER
9          LinEq::connect<1,double>(gaussElimination<double>);
10 #else
11         LinEq::connectID<1,double>(gaussElimination<double>);
12 #endif
13 }
```

Listing 7: Connecting the implementation

The CI (see listing 6) just defines the template function `solve()` for solving a system of linear equations of the form $Ax = b$. The function `CTL_connect()` (see listing 7) is also quite straightforward - the function is specialised for `double` and connected to the implementation function. The `ifdefs` are needed, because newer CTL versions (which incidentally define **CTL_VER**) have merged the former separate functions `connect()` and `connectID()`. The MEX wrapper code can be generated by running *ctlmex.py*:

```
$ ctlmex.py lineq.ci >lineq_mlab.cpp
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <mex.h>

#include <mexhelper.h>
#include "lineq.ci"

extern "C"
{
void mexFunction (int nresults, mxArray **arrresults, int nargs,
        const mxArray **arrargs)
{
        matrix<double> arg0 = matrix<double>();
        if (nargs > 0)
                mexConvert(arrargs[0], arg0);
        myVector<double> arg1 = myVector<double>(1);
        if (nargs > 1)
                mexConvert(arrargs[1], arg1);


        try
        {
                initEnv();
                myVector<double> res = LinEq::solve<double>(arg0,arg1);

                if (nresults > 0)
                {
                        mexConvert(res, arrresults[0]);
                }
        }
        catch (ctl::exception &e)
        {
                std::cerr << e << "\n";
        }
}
}
```

Listing 8: MEX wrapper code for the `LinEq` CI

The automatically generated code (see listing 8) is of course very similar to the one presented earlier, but uses the template function *mexConvert()* to do the conversions.

3.3 How to Run the MEX-File

Running a MEX-file is identical to using standard M-files, see listing 9. However, one needs to specify a location for the CTL component by calling `use()` before the desired function (more details in file *use.cpp* of the example code).

```
use('LinEq', './lineq -l tcp')
A = [4,1,0;1,4,1;0,1,4];
b = [0.5;0.5;0.5];
```

```
4 x = lineq_mlab(A, b);
5 x
```

Listing 9: Running the Gauss example

4 How to Use CTL4j Components from MATLAB

Using a CTL4j component from MATLAB is pretty straightforward, as MATLAB comes with support for calling Java code in M-files. However, generics are not supported. As a consequence the CTL4j static method `Location.parseFile()` cannot be used to read locations from a file (like in the CTL4j examples), as it returns a `LinkedList<String>`. Additionally, it might be necessary to modify MATLAB's CLASSPATH, so that it can find the CTL4j classes. Moreover, the MATLAB code calling the component looks like a standard CTL4j client application, see listing 10.

```
1 javaaddpath('/path/to/jsch-20041105-1.4.jar')
2 javaaddpath('/path/to/ctl4j/build')
3
4 loc = CTL.Types.Location('user@host:/path/to/ctl4j/src/Example.Server tcp')
5 proc = CTL.Process(loc)
6 javaSys.CTestCI.use(proc)
7 test = javaSys.CTestCI()
8 test.add(3, 4)
9 proc.stopService()
```

Listing 10: Calling CTL4j code from MATLAB

In contrast to a standard Java client, a created process is not stopped automatically. It needs to be done manually at the end of the MATLAB script by using `stopServices()`.

5 How to Realise a CTL/C++ Component from a MATLAB Code

The MATLAB language can be used to formulate solutions to mathematical problems faster and usually shorter than traditional programming languages. As written in chapter 2 making MATLAB code available to external applications involves multiple steps. Thus this can be a repetitive and error-prone task. Embedding MATLAB code inside a CTL component increases its accessibility, both in terms of usability as well as enabling distributed and parallel systems. This chapter presents how to do this manually and then introduces an automatic tool.

5.1 Manual Approach

In this section MATLAB's intrinsic function `eig()` is used. It is wrapped into the function `myeig()` from listing 11 to make it accessible for the MCC. Such a wrapping is only required for built-in functions.

```
function Y = myeig(X)
        Y = eig(X);
```

Listing 11: MATLAB function (*myeig.m*)

First of all, the shared library for `myeig()` is generated by the MCC:

```
$MCC_HOME/mcc -W cpplib:libmyeig -T link:lib myeig.m
```

This will produce several files, most importantly a shared library *libmyeig.so* and a header file *libmyeig.h*. From now on, the function can be used from C++ code, see listing 12.

```
#include "libmyeig.h"

int main (int argc, char **argv)
{
        libmyeigInitialize();
        double data[] = {1, 2, 3, 4};
        mwArray ret, arg(2, 2, mxDOUBLE_CLASS, mxREAL);
        arg.SetData(data, 4);
        myeig(1, ret, arg);
        std::cout << std::endl << "Eigenvalues:" << std::endl << ret << std::endl
            ;
        libmyeigTerminate();
        return 0;
}
```

Listing 12: Calling the function (*test.cpp*)

Before interfacing MATLAB, its runtime environment needs to be initialised using function `libmyeigInitialize()`. To clean up the environment the function `libmyeigTerminate()` is called. Of course, both functions will have the prefix given to MCC via the switch **-W**. So using '-W cpplib:foo' leads to a function `fooInitialize()`.

An extraction of *libmyeig.h* is shown in listing 13. A generated C function has a special calling convention. It returns void. The first argument is an integer `nargout` specifying the number of return values. These output values follow as a reference to `mwArray`. Finally the input arguments are specified as a constant reference to `mwArray`. Remember that MATLAB supports multiple return values. This is satisfied by `nargout` and the non-constant reference. Additionally, as in MATLAB, `nargout` can be smaller than the number of return values of the function, if the caller is only interested in the first outputs.

```
extern LIB_libmyeig_CPP_API void myeig(int nargout, mwArray& Y, const mwArray& X)
    ;
```

Listing 13: Header extraction for the generated code (*libmyeig.h*)

Typically one would convert the result in listing 12 to a standard C++ type, for example by using method `GetData()`, but this example just prints it to the screen.

To create a CTL component from the shared library we need to have a CI fitting to its functionality. As we want to use standard C++ types, a C++ wrapper function `my_eig()` is implemented doing the conversion and the communication with the shared library. Here, this is done inside the connect file, see listing 14. We define a corresponding CI containing function `Matlab::eig` which corresponds to our `my_eig()`, see listing 15.

```
#define CTL_Connect

#include <eig.ci>
#include <libmyeig.h>

std::vector<double> my_eig (std::vector<std::vector<double> > mat)
{
        libmyeigInitialize();

        double data[mat.size()*mat[0].size()];
        for (int i=0;i<mat.size();i++)
                for (int j=0;j<mat[i].size();j++)
                        data[i*j] = mat[i][j];
    mwArray ret, arg(mat.size(), mat[0].size(), mxDOUBLE_CLASS, mxREAL);
    arg.SetData(data, mat.size()*mat[0].size());

    myeig(1, ret, arg);

    mwArray arr = ret.GetDimensions();
    int size[2];
    arr.GetData(size, 2);

    double *d = (double *)malloc(sizeof(double)*size[0]);
    ret.GetData(d, size[0]);

    std::vector<double> ret2(size[0]);
    int k = 0;
    for (int i=0;i<size[0];i++)
                ret2[i] = d[k++];

    libmyeigTerminate();

    return ret2;
}

void CTL_connect ()
{
#ifdef CTL_VER
        Matlab::connect<1>(my_eig);
#else
        Matlab::connectID<1>(my_eig);
#endif
```

```
}
```

Listing 14: Connect (*eig_connect.cpp*)

```
#ifndef __EIG_CI__
#define __EIG_CI__

#include <ctl.h>

#define CTL_Library Matlab
#include CTL_LibBegin
        #define CTL_Function1 array<double>, eig, (const array<array<double> >),
            1
#include CTL_LibEnd

#endif
```

Listing 15: CI (*eig.ci*)

From all the code we compile and link a CTL service executable *eig.exe*. Listing 16 shows a client application calling the service.

```
#include <eig.ci>

int main (int argc, char **argv)
{
        std::vector<std::vector<double> > arg(2);
        arg[0].resize(2);
        arg[1].resize(2);
        arg[0][0] = 1; arg[0][1] = 2;
        arg[1][0] = 3; arg[1][1] = 4;
        std::vector<double> res;

        try
        {
                Matlab::use(ctl::link("./eig.exe -l tcp"));
                res = Matlab::eig(arg);
        }
        catch (ctl::exception &e)
        {
                std::cerr << "Error: " << e << "\n";
        }

        for (int i=0;i<res.size();i++)
                std::cout << "Eigenvalue: " << res[i] << "\n";
        return 0;
}
```

Listing 16: Client for calling service *eig.exe* (*eig_client.cpp*)

Notice, that before running the application some environmental variables need to be set for the MCR. When running a component remotely, one can embed the environmental variable settings and the service invocation subsequently inside a shell script. Instead of calling *eig.exe*, one needs to call the shell script. Alternatively the settings can be avoided by linking to the specific MCR required shared libraries, see section 5.2.

Contrary to the upper example, the component can be realised using a C++ class. The class initialises the shared library typically inside the constructor and terminates it inside the destructor. Now we want to switch to the case, in which such a component is instantiated more than one time locally on a machine. Thus multiple running components interface the same MATLAB code. In this case we have to be careful with the initialisation and termination of the shared library. As the first instantiated component need to initialise the library and the last need to terminate the library, we introduced a reference counter. This counter is initialised globally with zero, incremented inside the constructor and decremented inside the destructor. If the counter is zero, when the constructor is called, the library is initialised. If the counter is zero after decrementing it in the destructor, the library is terminated.

5.2 Automatic Approach

The last section showed the somehow time-consuming manual approach. In the process of writing this paper, a code-generator was developed which automates the work and provides a CI for one MATLAB function.

The code generation is done by two Python scripts *gen_makefiles.py* and *dotm2cpp.py*. *gen_makefiles.py* generates the necessary Makefiles. *dotm2cpp.py* does the code generation and conversion by including *matlab.h*. This header defines the class `MyMwArray` derived from `mwArray`, which handles type conversions of itself to `double`, `int`, `std::string`, `std::vector<double>` and `std::vector<std::vector<double> >` and vice versa. After parsing the M-file, the following files are generated:

- A thin C++ wrapper around the code generated by the MCC,
- a functional CI specifying the function from the MATLAB script,
- the necessary `CTL_connect()` code, which is needed to compile a working component.

After compiling and linking it to the shared library (compiled by the MCC), a fully-functional CTL/C++ component results, which calls the function written in MATLAB.

The thin C++ wrapper function is a template, because MATLAB is a dynamically typed language. Of course the function needs to be instantiated for every permutation of the argument and return types. A component for a function with more than three arguments and return values results in 243 instantiations. This leads to a lot of generated code and takes perhaps a long time to compile. If a function with more arguments is needed, one can specify the types manually inside the M-file, see listing 17.

```
function [x1,flag1,rr1,iter1,rv1] = mypcg2(A, b)
tol = 1e-6;  maxit = 15;  M = diag([10:-1:1 1 1:10]);
[x1, flag1, rr1, iter1, rv1] = pcg(A, b, tol, maxit, M);
% @types vector, double, double, double, vector, matrix, vector
```

Listing 17: Example for manually specifying types

More recent versions of the MCC might not contain the bug described in chapter 2.3 and thus does not need the workaround provided by the automatism. It can be disabled by setting the flag `has_bug` in *matlab.h* to `false`.

5.2.1 Example 1: Eigenvalues

In this example, we consider `myeig.m` from the manual approach. The Makefile generated by *gen_makefiles.py* automates all the steps for building the component. Thus invoking *make* is the only thing one needs to do. In listing 18 a client application is presented. The template function is embedded into the namespace `Matlab`. It needs to be specialised in the order: type of the first return value, types of the arguments, types of possible additional return values). `std::vector<std::vector<double> >` is used for both the return value and the argument. All conversions from native C++ to the MATLAB types are done transparently inside the generated code. Basically, everything works like a namespace `Matlab` with one function `myeig()`. The header file *client.h* just defines an alias for the matrix type and a pretty-printer for it, see listing 19.

```
#include <mlab_myeig.ci>
#include <client.h>

int main (int argc, char **argv)
{
        ctl::vector<ctl::location> loc = ctl::readLocation("locs.txt");
        if (loc.size() < 1)
        {
                std::cerr << "No valid location found.\n";
                return 1;
        }

        for(int P=0; P<loc.size(); P++)
        {
                try
                {
                        ctl::link lnk(loc[P]);
                        Matlab::use(lnk);
                        myMatrix arg(2), res;
                        for (int i=0;i<2;i++)
                                arg[i].resize(2);
                        arg[0][0] = 1;
                        arg[0][1] = 2;
                        arg[1][0] = 3;
                        arg[1][1] = 4;
                        res = Matlab::myeig<myMatrix,myMatrix>(arg);
```

```
27                          std::cout << "Eigenvalues:\n" << res << "\n";
28                  }
29                  catch (ctl::exception &e)
30                  {
31                          std::cerr << e << "\n";
32                  }
33          }
34
35          return 0;
36  }
```

Listing 18: Example CTL/C++ client (*client.cpp*)

```
1   #ifndef __CLIENT_H__
2   #define __CLIENT_H__
3
4   #include <sys/stat.h>
5
6   #include <vector>
7
8   typedef std::vector<std::vector<double> > myMatrix;
9   typedef std::vector<double> myVector;
10  //typedef std::vector<int> myVector;
11
12  std::ostream& operator<<(std::ostream& os, const myVector &v)
13  {
14          for (int i=0;i<v.size();i++)
15          {
16                          os << "\t";
17                          if (v[i] >= 0)
18                                  os << " ";
19                          os << v[i];
20                          if (i!=v.size()-1)
21                                  os << "\n";
22          }
23          return os;
24  }
25
26  std::ostream& operator<<(std::ostream& os, const myMatrix &m)
27  {
28          for (int i=0;i<m.size();i++)
29          {
30                  for (int j=0;j<m[i].size();j++)
31                  {
32                          os << "\t";
33                          if (m[i][j] >= 0)
34                                  os << " ";
35                          os << m[i][j];
36                  }
37
38                  if (i!=m.size()-1)
39                          os << "\n";
40          }
41          return os;
42  }
43
44  #endif
```

Listing 19: Example CTL/C++ client (*client.h*)

5.2.2 Example 2: PCG

This section shows a more complex example, which uses two MATLAB functions. `getproblem()` returns a coefficient matrix A and a right-hand side b of the problem $Ax = b$. `mypcg()` solves the system using MATLAB's Preconditioned Conjugate Gradient (PCG) solver. The two M-files are shown in the listings 20 and 21, a client in listing 22.

```
function [A, b] = getproblem(n)
        A = gallery('moler', n);
        b = A*ones(n, 1);
```

Listing 20: Example function for generating a problem

```
function x1 = mypcg(A, b)
        x1 = pcg(A, b);

%function [x1,flag1,rr1,iter1,rv1] = mypcg(A, b)

%tol = 1e-6;  maxit = 15;  M = diag([10:-1:1 1 1:10]);
%[x1, flag1, rr1, iter1, rv1] = pcg(A, b, tol, maxit, M);
```

Listing 21: Example PCG solver

To avoid a conflict between two `Matlab` namespaces, both Python scripts support the command line option *-s*. *-sFOO* results in the suffix **FOO** to the namespace, so it becomes *MatlabFOO*.

```
#include <mlab_getproblem.ci>
#include <mlab_mypcg.ci>
#include <client.h>

int main (int argc, char **argv)
{
        const char *loc0 = "mlab_getproblem.exe", *loc1 = "mlab_mypcg.exe";
        ctl::link lnk0(loc0), lnk1(loc1);
        try
        {
                myMatrix A;
                myVector b;

                Matlabprob::use(lnk0);
                A = Matlabprob::getproblem<myMatrix, int, myVector>(21, b);

                Matlab::use(lnk1);
                myVector x = Matlab::mypcg<myVector, myMatrix, myVector>(A, b);
                std::cout << "Result obtained by pcg():\n" << x << "\n";
        }
        catch (ctl::exception &e)
        {
                std::cerr << e << "\n";
        }
        return 0;
}
```

Listing 22: Example CTL/C++ client (*client2.cpp*)

5.2.3 Example 3: Eval

The third MATLAB example is presented in listing 23. Function `myeval()` gets function values `xs` and a function name as a string `funid`. Inside, the specified input function is called for each of the above function values. This is achieved by using MATLAB's function `eval()`, which evaluates a string as if it was typed on the MATLAB console. This solution serves as a kind of dynamic function call.

```
1  function myeval(xs, funid)
2          ys = [];
3          for x=xs
4                  str = ['ys = [ys, ', funid, '(', sprintf('%i', x), ')];'];
5                  eval(str);
6          end;
7
8          xlabel('x');
9          ylabel('y');
10         plot(xs, ys);
11         print('-depsc', 'eval-out.eps')
12 % @types void, vector, string
```

Listing 23: Example evaluate function (*myeval.m*)

One could wonder about the weird string concatenation instead of just using `sprintf()`. A straightforward way would result in listing 24. Unfortunately, this code only works when running inside MATLAB, but fails, when it is used in a shared library compiled by MCC. Therefore the concatenation workaround is taken.

```
1 str = sprintf('ys = [ys, %s(%i)];', funid, x);
```

Listing 24: MATLAB `sprintf()`

We consider `foo()` as a function which is passed to `myeval()`, see listing 25.

```
1 function y = foo(x)
2         y = x^2;
```

Listing 25: Example function being evaluated (*foo.m*)

In the manual approach one can give the MCC the two M-files directly, see also chapter 2.3. Contrary, in the automatic approach the MCC is called two times for generating a shared library for each function. The client application needs to know about the second function too and initialise it properly by calling `libfooInitialize()`. It also needs to copy the resulting pre-compiled *foo.m* to the MCR directory of *libmyeval*, so that it can be found at runtime. To hide this from the user, *dotm2cpp.py* provides the command line switch *-u*. It generates the corresponding Makefiles for each function, inserts some initialisation code into the component and copies the pre-compiled M-file of the second function to

the target directory of the first one. Summarised, one can call the second function from the first one without manually writing any C++ support code, but the Makefile generated for the second one has to be called before the one for the first.

Listing 26 shows a client for the generated component. Unfortunately, MATLAB's plotting support does not work as expected from the wrapper library generated by MCC. Therefore an empty plot will be produced.

```
#include <mlab_myeval.ci>
#include <client.h>

/* This needs to be run twice to have foo.m in the right directory. */

int main (int argc, char **argv)
{
        const char *loc = "mlab_myeval.exe";
        ctl::link lnk(loc);
        try
        {
                myVector xs(5);
                for (int i=0;i<xs.size();i++)
                        xs[i] = i+1;
                std::string funid("foo");

                Matlab::use(lnk);
                unlink("eval-out.eps");
                Matlab::myeval<void, myVector, std::string>(xs, funid);
                sleep(4);
                int foo = access("eval-out.eps", R_OK);
                std::cout << foo << "\n";
        }
        catch (ctl::exception &e)
        {
                std::cerr << e << "\n";
        }
        return 0;
}
```

Listing 26: Example CTL/C++ client (*client3.cpp*)

Notice, if a void MATLAB function is used, the code generator will still generate a wrapper function with a return value. So one has to pass some dummy template argument to it in the client code (here, we take `void`). As the special case of defining a void function is rarely used, it is not handled explicitly.

5.2.4 Remarks for Remote Access

As described earlier, MATLAB's runtime needs a specific environment. However, the code-generator handles all the settings automatically, so that the resulting executables can be used directly, just like any other CTL component.

5.2.5 Concurrency

The initialisation process of MATLAB's runtime causes problems when accessing MCC-compiled code concurrently. As the automatic approach generates CTL service executables, the above problem can be avoided by using multiple `ctl::link` objects, see listing 27. In the example, the same MATLAB function `fun2()` is used as in section 5.3.

```
#include <mlab_fun2.ci>

int main (int argc, char **argv)
{
        ctl::vector<ctl::location> loc = ctl::readLocation("locs.txt");
        if (loc.size() < 1)
        {
                std::cerr << "No valid location found.\n";
                return 1;
        }

        try
        {
                ctl::link lnk(loc[0]);
                ctl::link lnk2(loc[0]);

                Matlab::use(lnk);
                std::cout << (double)Matlab::fun2<double, int, int>(5, 6) << "\n"
                    ;

                // Using a second link is required for calling the same function
                    twice
                Matlab::use(lnk2);
                std::cout << (double)Matlab::fun2<double, int, int>(5, 6) << "\n"
                    ;

                // Concurrency in one process needs two links (this does NOT work
                     for
                // shared objects!)
                ctl::link lnk3(loc[0]);
                ctl::link lnk4(loc[0]);
                ctl::result<double> res1 = Matlab::fun2<double, int, int>(lnk3,
                    1, 2);
                ctl::result<double> res2 = Matlab::fun2<double, int, int>(lnk4,
                    3, 4);
                std::cout << "Results: " << (double)res2 << ", " << (double)res1
                    << "\n";
        }
        catch (ctl::exception &e)
        {
                std::cerr << e << "\n";
        }

        return 0;
}
```

Listing 27: Example for accessing CTL/MATLAB components remotely

5.2.6 Limitations

Some limitations are present in the automatic approach.

- All vectors are treated as column vectors.
- Concurrent access is not always possible.
- Plotting is not working correctly.

5.3 How to Use the Component from CTL4j

We show a very simple MATLAB function (see listing 28), which is called from a CTL4j client by using an auto-generated CTL/C++ component (see CI in listing 29). The reason for the simplicity lies in the fact, that templated matrix and vector types are not yet available in the CTL4j. Therefore the intersection between Java and MATLAB lies only in integer and floating point fundamentals.

```
function y = fun2(x1, x2)
        y = x1*x2;
```

Listing 28: Simple MATLAB function

```
#include <ctl.h>
#ifndef __MLAB_FUN2_H__
#define __MLAB_FUN2_H__

#define CTL_Library Matlab
#include CTL_LibBegin
#define CTL_FunctionTmpl1 RET, fun2, (const X1x1, const X2x2), 2, (RET, X1x1,
    X2x2), 3
#include CTL_LibEnd

#endif // __MLAB_FUN2_H__

// vim: ft=cpp
```

Listing 29: Generated CI for the simple MATLAB function

Using a CTL/MATLAB component works similar as accessing any other CTL/C++ component from CTL4j code. The following steps are required. More details onto the steps are given in the project work [1].

- Add a new block to the 'genri' target in *build.xml*, see listing 30.

```
<exec executable="${ctlcc}" logError="true"
        output="${src}/_default/MatlabCTLI.java">
        <arg value="ctlcc.py"/>
        <arg value="-F"/>
        <arg value="-q"/>
        <arg value="-DMatlab=MatlabCTLI"/>
        <arg value="matlab/cpp/codegen/mlab_fun2.ci"/>
```

```
</exec>
```

Listing 30: Ant buildfile block 1

The code generator *ctlcc.py* generates Java code from a CI. Of course the namespace (here, `Matlab`) specified in the CI has to be used. The suffix `CTLI` informs the CTL4j code generator, that just a remote interface is generated and none of the other possible wrappers. The switch '-F' specifies, that the CI is strictly functional and '-q' deactivates the warnings for undefined template types. The directive '-D' is always needed for classes with the `CTLI` suffix. The final parameter is the path to the CI.

- Now another block is added to the 'compile-gen' target, see listing 31.

```
<java classname="CodeGen.Main" classpath="${classpath}" fork="true">
        <arg value="-DRET=int"/>
        <arg value="-DX1x1=int"/>
        <arg value="-DX2x2=int"/>
        <arg value="_default.MatlabCTLI"/>
</java>
```

Listing 31: Ant buildfile block 2

This calls the CTL4j code generator for the Java class created earlier. For each template parameter in the CI, a '-D' directive has to be added to specialise the type, as this cannot be done in Java itself. As written above, MATLAB is a dynamically typed language. This is reflected by the CTL/C++ components by using template functions. In the Java case the type parameters are defined statically. Thus certain functions might be more restricted in Java applications.

- Finally, the client code can be implemented, see listing 32.

```
import _default.*;

public class Client
{
        public static void main (String[] args)
        {
                for (CTL.Types.Location loc : CTL.Types.Location.parseFile(
                    "locs.txt"))
                {
                        CTL.Process proc = new CTL.Process(loc);
                        Matlab.use(proc);
                        Matlab tmp = new Matlab();
                        System.out.println(tmp.fun2(2, 2));
                }
        }
}
```

Listing 32: Example Java client

As in the C++ case, this looks just like any other CTL4j client. Before running it, the file *locs.txt* has to be altered to point to the correct service executable.

- The output of the example is shown in the following.

```
HOST:~/projects/wire/ctl4j$ ./run.sh
4
Warning: Unable to open display , MATLAB is starting without
a display.
You will not be able to display graphics on the screen.
Warning: No window system found.  Java option 'MWT' ignored.
Warning: Unable to load Java Runtime Environment: libjvm.so:
cannot open shared object file: No such file or directory.
Warning: Disabling Java support.
```

Some warnings from MATLAB occur, as well as the calculation result.

6 Conclusion

In this paper, we presented solutions on coupling C++ or Java code with MATLAB, utilising the CTL to enable everything in a distributed hybrid software system. Therefor we used the MCC and MEX offered by MATLAB. In addition to the manual approaches, we provide two code-generators. They automate the coupling process. Both generators are released under the terms of the *GNU General Public License* (*GPL*) version 2. To simplify the manual approach we implemented a C++ class `mwArray_prs`, which does the type conversion, divides between row- and column vectors and handles the MCC bug described in chapter 2.3. This class is part of a bigger package, which is planned to be available on *SourceForge* in the near future.

The example code is available under `https://shuya.ath.cx/~neocool/tmp/matlab/sample.tar.gz`. Further development of the code generators happens inside the CTL4j SVN repository located at `https://shuya.ath.cx/svn-legacy/ctl4j/`.

Appendix: Development Environment and Installation

All of the example code presented in this paper was built and tested on Linux (Arch Linux 0.8) with

- GCC 3.3.6,
- CTL4j version 0.9.7,
- CTL/C++ version 1.2.1,
- MATLAB version 7.2.

Important: The CTL/C++ has to be compiled with GCC 3.3.x. Also the *g++* in *PATH* needs to be of the same version 3.3.x. Otherwise the linker will have two or more different shared libraries *libstdc++* to choose from, which leads to undefined runtime behaviour. Even though GCC 3.3.x is the only compiler supported by the code-generators, one may want to use a different one, using the following procedure:

```
cd $MATLAB
cd sys/os/glnx86
mkdir old
mv libstdc++.* libg2c.* libgcc_s* old
```

Now the corresponding libraries of the GCC installation are copied into the directory *sys/os/glnx86/* [2].

If the administrator did not install GCC 3.3.x or the installed distribution does not provide it anymore, one can also compile it very easily. After the GCC tarball (*gcc-3.3.6.tar.bz2*) is downloaded from the local GNU mirror, the following commands can be used to compile and install it in the home directory (∼50MB of free disk space required):

```
mkdir gcc-build &&
cd gcc-build &&
../gcc-3.3.6/configure --prefix=$HOME/software/gcc-3.3
--enable-shared --enable-languages=c,c++ \
--enable-threads=posix --enable-__cxa_atexit \
--enable-clocale=gnu &&
make bootstrap &&
make install
```

This GCC can be used instead of the system's one by putting *$HOME/software/gcc-3.3/bin* in front of the **PATH**.

References

[1] Boris Bügling. The CTL protocol and its Java implementation. http://www.wire.tu-bs.de/forschung/projekte/ctl/files/ctl_spec.pdf, 2006.

[2] MathWorks. Using a newer GCC with MATLAB. http://www.mathworks.com/support/solutions/data/1-QBCS1.html?solution=1-QBCS1.

[3] MathWorks. Working with the MCR. http://www.mathworks.com/access/helpdesk/help/toolbox/compiler/index.html?/access/helpdesk/help/toolbox/compiler/f12-999353.html&http://www.google.com/search?q=matlab%20mcr&hl=en&client=ig.

[4] Dr. Rainer Niekamp. CTL Manual for Linux/Unix. http://www.wire.tu-bs.de/forschung/projekte/ctl/files/manual.pdf, 2005.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2003-04

2003-04	A. Keese, H. G. Matthies	Fragen der numerischen Integration bei stochastischen finiten Elementen für nichtlineare Probleme
2003-05	A. Keese, H. G. Matthies	Numerical Methods and Smolyak Quadrature for Nonlinear Stochastic Partial Differential Equations
2003-06	A. Keese	A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements)
2003-07	M. Meyer, H. G. Matthies	State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics
2003-08	H. G. Matthies, A. Keese	Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations
2003-09	A. Keese, H. G. Matthies	Parallel Computation of Stochastic Groundwater Flow
2003-10	M. Mutz, M. Huhn	Automated Statechart Analysis for User-defined Design Rules
2004-01	T.-P. Fries, H. G. Matthies	A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods
2004-02	B. Mathiak, S. Eckstein	Automatische Lernverfahren zur Analyse von biomedizinischer Literatur
2005-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme
2005-02	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization
2005-03	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling
2005-04	H. Krahn, B. Rumpe	Evolution von Software-Architekturen
2005-05	O. Kayser-Herold, H. G. Matthies	Least-Squares FEM, Literature Review
2005-06	T. Mücke, U. Goltz	Single Run Coverage Criteria subsume EX-Weak Mutation Coverage
2005-07	T. Mücke, M. Huhn	Minimizing Test Execution Time During Test Generation
2005-08	B. Florentz, M. Huhn	A Metamodel for Architecture Evaluation
2006-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme
2006-02	T. Mücke, B. Florentz, C. Diefer	Generating Interpreters from Elementary Syntax and Semantics Descriptions
2006-03	B. Gajanovic, B. Rumpe	Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen
2006-04	H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel	Handbuch zu MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen
2007-01	M. Conrad, H. Giese, B. Rumpe, B. Schätz (Hrsg.)	Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III
2007-02	J. Rang	Design of DIRK schemes for solving the Navier-Stokes-equations
2007-03	B. Bügling, M. Krosche	Coupling the CTL and MATLAB